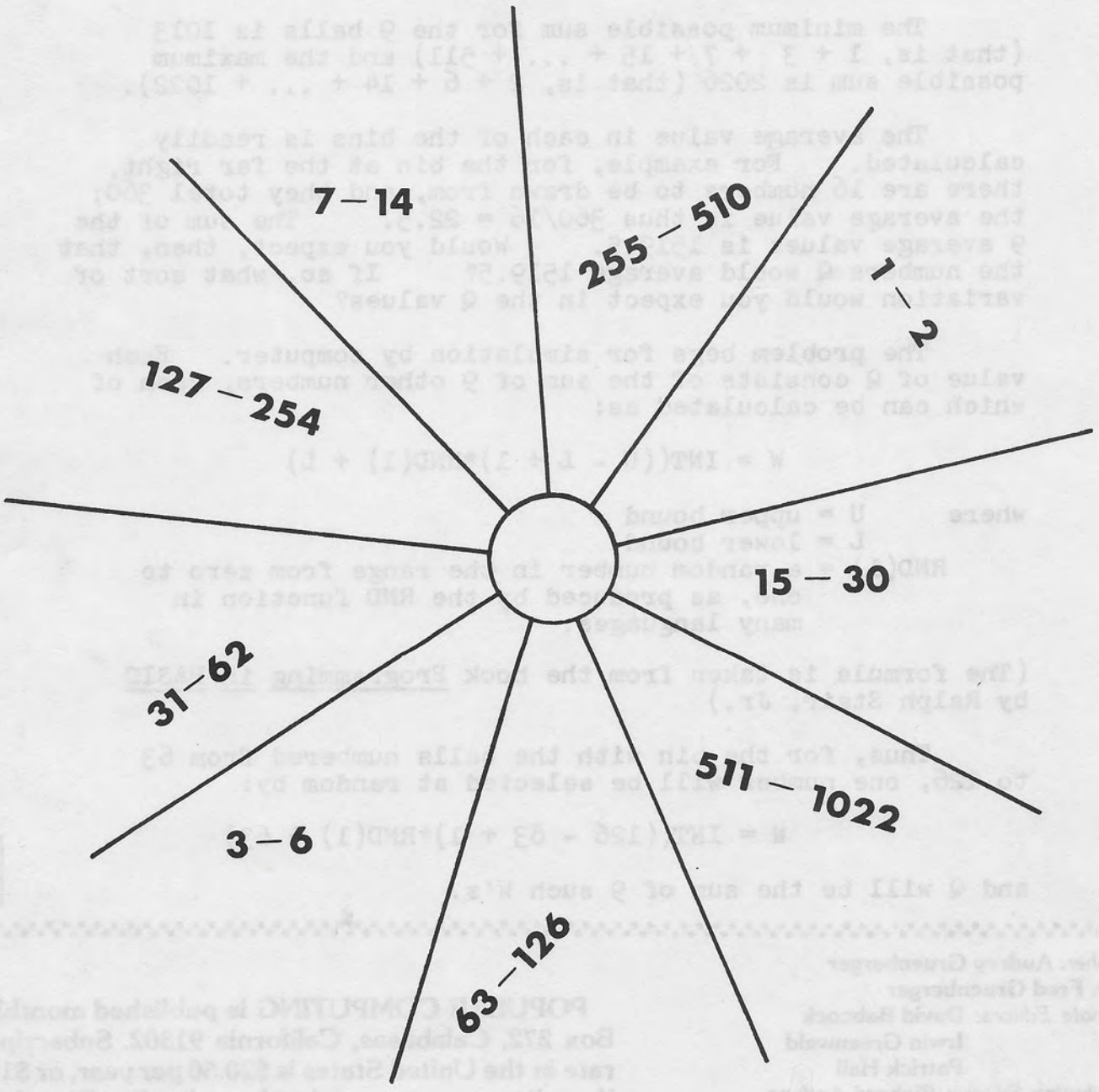
Popular Computing

79

The world's only magazine devoted to the art of computing.

October 1979 Volume 7 Number 10



Nine Bins

So you think you understand how random processes will behave?

The diagram on the cover represents 9 bins, each of which contains an unlimited number of balls, numbered uniformly through the indicated range. Thus, the bin at the bottom contains infinitely many balls numbered in equal quantities from 63 to 126.

A ball is drawn at random from each bin, and the sum of the 9 numbers constitutes one play; call the sum Q.

What would you expect for the distribution of the Q values?

The minimum possible sum for the 9 balls is 1013 (that is, 1 + 3 + 7 + 15 + ... + 511) and the maximum possible sum is 2026 (that is, 2 + 6 + 14 + ... + 1022).

The average value in each of the bins is readily calculated. For example, for the bin at the far right, there are 16 numbers to be drawn from, and they total 360; the average value is thus 360/16 = 22.5. The sum of the 9 average values is 1519.5. Would you expect, then, that the numbers Q would average 1519.5? If so, what sort of variation would you expect in the Q values?

The problem begs for simulation by computer. Each value of Q consists of the sum of 9 other numbers, each of which can be calculated as:

```
W = INT((U - L + 1)*RND(1) + L)
```

where U = upper bound
L = lower bound
RND(1) = a random number in the range from zero to one, as produced by the RND function in many languages.

(The formula is taken from the book Programming in BASIC by Ralph Stair, Jr.)

Thus, for the bin with the balls numbered from 63 to 126, one number will be selected at random by:

```
W = INT((126 - 63 + 1)*RND(1) + 63)
```

and Q will be the sum of 9 such W's.

Publisher: **Audrey Gruenberger**
Editor: **Fred Gruenberger**
Associate Editors: **David Babcock, Irwin Greenwald, Patrick Hall**
Contributing Editors: **Richard Andree, William C. McGee, Thomas R. Parkin, Edward Ryan**
Art Director: **John G. Scott**
Business Manager: **Ben Moore**

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year. For all other countries, add $3.50 per year. Back issues $2.50 each.

3X+1 Revisited

The 3X+1 Problem was the opening article in our first issue, and has reappeared in issues 4, 13, 25, and 45. It seems to be time to consider Prof. Andree's fascinating problem once again.

For any positive integer N, we let X = N and then follow this algorithm:

Replace X with 3X+1 if X is odd;

Replace X with X/2 if X is even;

Repeat until X = 1; and

Let A = the number of terms so generated.

The process is illustrated completely for N = 9 on another page.

The simple algorithm leads immediately to some interesting problems:

1. Can it be proved that the algorithm will converge to <u>one</u> for any value of N? It certainly seems to do so. The convergence has been verified for all values of N up to 222,000,000 by computer runs. A flowchart for such a run is given, Figure E. Three things are noteworthy:

A. If the validation is made systematically, taking successive values of N, then only <u>odd</u> values of N need to be tested. Any even value goes immediately to N/2, which has already been tested.

B. The procedure need only proceed to the point at which X < N, for the same reason. In the illustration for N = 9 (Figure F), X becomes 7 on the fourth line, and the test can stop there, since 7 has already been tested.

C. Numbers of the form 4K+1 can be shown to converge analytically:

N = 4K+1 (odd)
12K+4 (even)
6K+2 (even)
3K+1 -- definitely less than N, and hence convergent by rule B above.

N = 9 = X, which is odd; replace with

28 which is even, so replace with

14

7

22

11

34

17

52

26

13

40

20

10

5

16

8

4

2

1

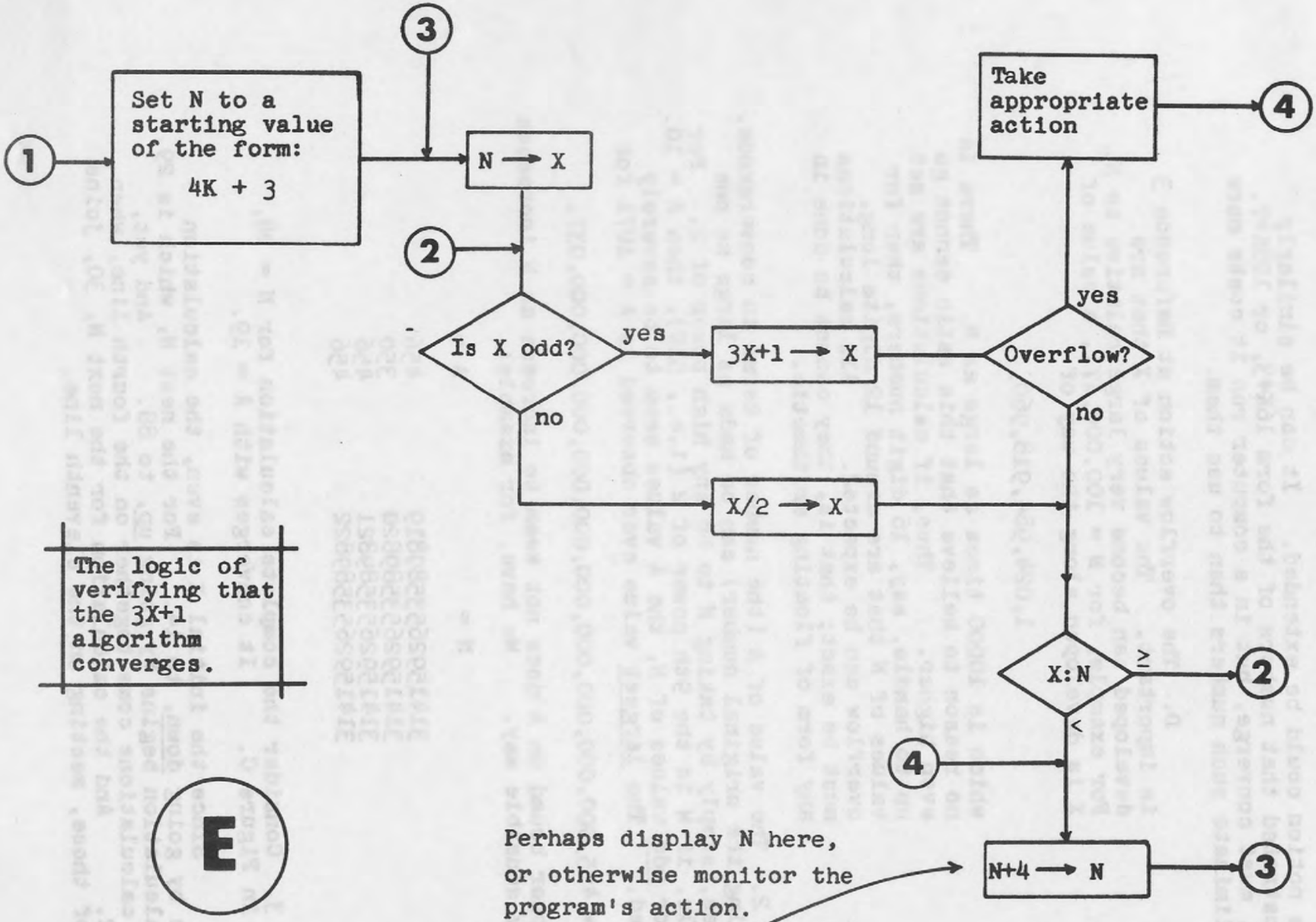
and the number of terms, including the original value, is A = 20.

First appearance of a string of length L. For example, a string of length 13 ends with the number N = 3994 (which means that it starts with N = 3982).

L	Ending N
2	13
3	30
4	317
5	102
6	391
7	949
8	1501
9	1688
10	4731
11	6586
12	11707
13	3994
14	3000
15	17562
16	36223
17	7099
18	59709
19	159134
20	79611
21	57877
22	212181
23	352280
24	221208
25	57370
26	294938
27	252574
28	530079
29	331806
30	524318
40	596349

F

The 3X+1 algorithm illustrated with the complete expansion for N = 9.



The logic of verifying that the 3X+1 algorithm converges.

E

Therefore, a validation run should always begin with an N value of the form 4K+3 (that is, 7, or 11, or 14, or 222,000,003, etc.) and test only every 4th N after that.

This notion could be extended. It can be similarly demonstrated that numbers of the form 16K+3, or 128K+7, etc., must converge, but in a computer run it costs more to eliminate such numbers than to use them.

D. The overflow action at Reference 3 is important. The values of X that are developed can become very large relative to N. For example, for N = 100,000,171, a value of X is developed along the way of

1,024,984,918,960,

which is 10000 times as large as N. There is no reason to believe that this ratio cannot go even higher. Thus, if calculations are set up to handle, say, 16 digit numbers, then for values of N that are around 12 digits long, overflow can be expected. All calculations must be exact; that is, they cannot be done in any form of floating arithmetic.

2. The value of A (the number of terms to convergence, counting the original number) can be made as large as one pleases, simply by taking N to be any high power of 2. For example, if N is the 9th power of 2 (i.e., 512), then A = 10. But for <u>odd</u> values of N, the A values seem to be severely limited. The <u>largest</u> value ever observed is A = 1071 for

N = 495,000,000,000,000,000,000,000,000,000,000,037.

The upper bound on A does not seem to increase as N increases in any sensible way. We have, for example:

N =	A =
3141592653589819	456
3141592653589820	350
3141592653589821	456
3141592653589822	456

3. Consider the complete calculation for N = 28, shown in Figure C. It converges with A = 19.

Since the initial X is even, the calculation begins by going <u>down</u>, to 14. For the next N, which is 29, the calculation begins by going <u>up</u>, to 88. And yet, those calculations come together on the fourth line, when X = 22. And the calculation for the next N, 30, joins both of those, meeting on the eleventh line.

28	29	30
14	88	15
7	44	46
22 ←	22	23
11		70
34		35
17		106
52		53
26		160
13		80
40 ←		40
20		
10		
5		
16		
8		
4		
2		
1		

The "string" effect on consecutive values of N.

Thus, three consecutive values of N have the same A value (and, indeed, go out in exactly the same way). These strings of equal A values reappear constantly. We have twice published tables of such strings, but both tables were in serious error. The new table, Figure F, is much better. The logic of sifting out such strings was given in a flowchart in issue 38.

4. But the problem of finding strings of equal A values can be extended to a more interesting situation, illustrated in Table D, which summarizes this information:

N	A
1 000 000 000	101
1 000 000 001	163
1 000 000 002	163
1 000 000 003	163
. . . . .	. .
. . . . .	. .
1 000 000 028	101

128

The circled 225 in Table D interrupts two strings of 362's, one of length 14, the other of length 7. Thus, there is a string of length 21, with one interruption. Problem: given a succession of A values (namely, the output of the 3X+1 algorithm for consecutive values of N), how could one detect and measure such broken strings?

5. It was pointed out that for even values of N, A could be made any size, but for odd values of N, A seems to be limited. Within that constraint, though, the inverse problem is unsolved: for a given value of A, is there always some odd value of N that will yield that value of A?

Further investigation into the "string" effect of the 3X+1 algorithm was conducted at N = one billion. In the following table, the end value of N is given for strings of length L; add one billion to the N values shown.

L	N
3	51
4	127
5	47
6	79
7	7
8	63
9	664
10	405
11	1226
12	1375
13	504
14	31
15	440
16	5931
17	8090
18	23800
19	19558
20	29439
21	1464
22	87349
23	22040
24	1560
25	12824
26	49306
27	55786
29	41918
30	6174
31	24664
32	200481
42	222165
44	29418
51	49373

[The last entry here establishes a record; it is the longest string so far observed. The 51 numbers from 1,000,049,323 through 1,000,049,373 all converge in 163 terms.]

101	163	163	163	163	163	163	163	256	163
256	163	256	256	349	349	362	256	362	362
362	362	362	362	362	362	362	362	362	362
362	362	225	362	362	362	362	362	362	362
225	362	225	362	362	362	362	362	225	362
362	362	225	225	207	207	362	362	362	362
362	362	362	362	225	256	256	256	163	163
163	163	163	362	163	163	163	163	163	163
225	362	362	362	225	225	163	163	225	362
225	163	225	225	362	362	225	163	225	225
225	225	225	163	225	163	225	362	207	207
207	362	163	225	300	300	163	163	225	225
163	300	163	225	362	362	362	362	101	

D

The 3X+1 problem has not yet been exhausted. Following is a list of possible remaining work:

1. Simple convergence (that is, the logic of Flowchart E) could be extended from 222,000,000, which was the last verified value for N. As was pointed out, a program to do that would have to be able to handle integers of at least 14 digits. It should be noted, though, that the algorithm has been applied to numbers over a tremendous range (at least up to 36-digit numbers) and has invariably converged.

2. Table F (the first appearance of longer lengths of strings) should be extended.

3. After the logic of detecting broken strings is worked out, then that logic should be applied to consecutive values of N.

4. The possible limits of the ratio:

$$\frac{\text{largest X}}{\text{N}}$$

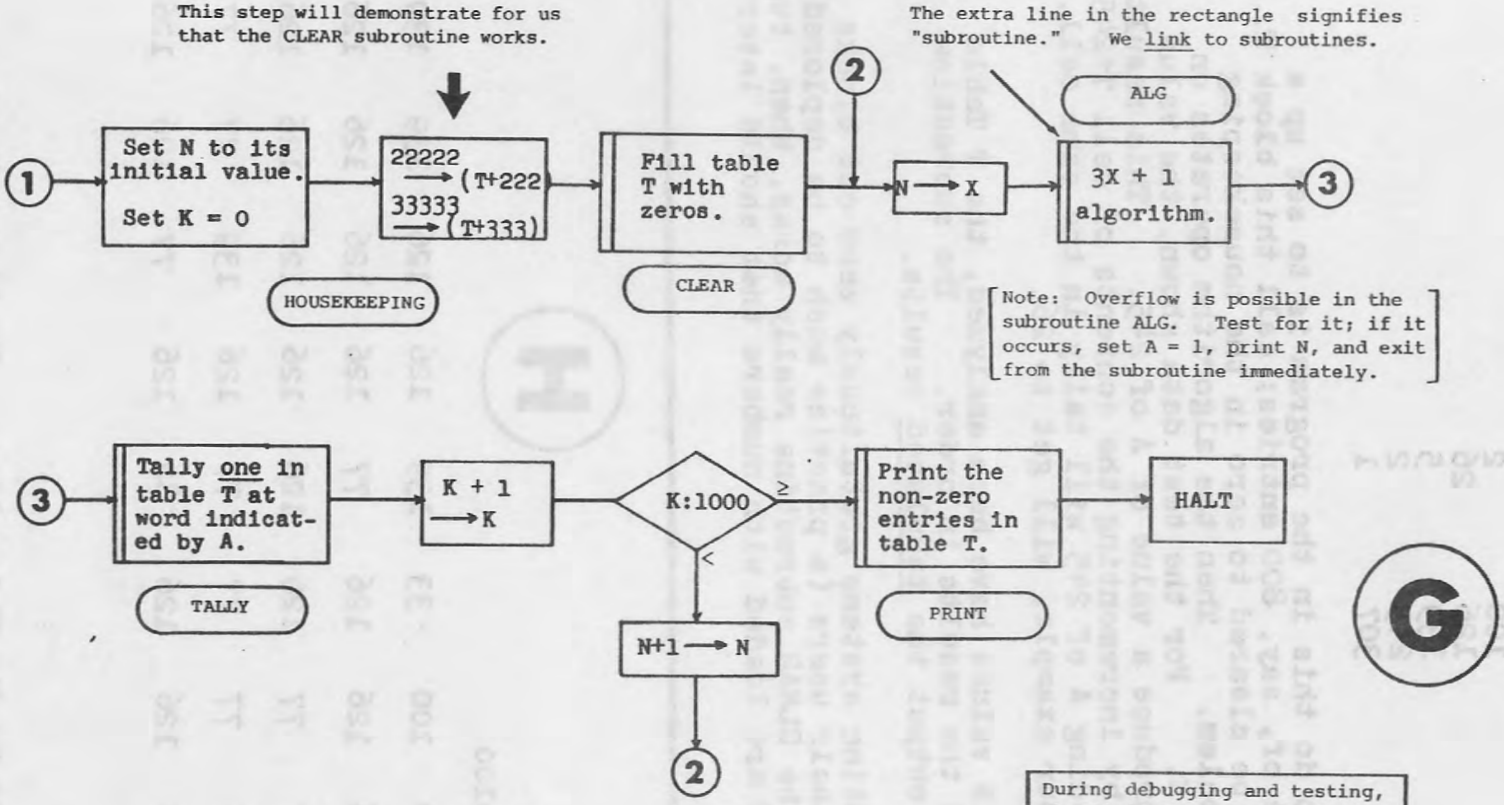
that can develop in the process of converging should be explored further. The ratio cited earlier (10249.8) is the highest so far observed, but there has been no systematic search for higher values of the ratio.

5. An A value greater than 1071 (for odd values of N) can surely be found, simply by handling larger values of N than have been so far calculated. What needs to be investigated is the rate of growth of A in relation to N. This would probably be the most valuable piece of research that could be done. A concerted attack, fleshing out a table like the one below, might shed some light on the mysterious growth of the A values.

N range (10 to the power:)	Largest A in this range:	Average A in this range:
16	517	405
25	790	647
31	947	748
35	975	651

The 3X+1 problem makes an excellent programming assignment for the last weeks of an introductory course. Each student can be shown how to write a program to calculate the A values for, say, 1000 consecutive values of N. This program will eventually execute some 20,000,000 instructions and hence offers some sense of achievement on a non-trivial problem. The programming can be done in assembly language, or in integer arithmetic in many high-level languages.

The problem is readily explained, and the calculations are easily carried out by hand for small values of N. (Each student should carry out the calculation for N = 27 by hand; it goes to 112 terms before converging.) Best of all, the overall problem lends itself to segmentation into five logical parts, as shown in Figure G.



3X+1

The test data furnished to you shows the A values for 52 consecutive values of N. Your program is to summarize (that is, tabulate) the test data, to produce only a few lines of output. Tabulate your test data by hand before making any runs on the machine.

Test data can be furnished (carefully selected to include some values that will overflow in the programs that the student writes). For the test data shown in Figure H, what is wanted for output is a tabulation:

33	1
77	14
82	1
100	2
126	26
139	5
245	2
307	1

One way to do this in the program is to set up a block of storage of, say, 800 entries; call this block T. The block is to be cleared to zero in the housekeeping phase of the problem. Then the algorithm operates on each successive N. For the test data shown, the value N = 79100 will produce a value of A of 245. This result will be tallied by incrementing the contents of cell T+245. Then each succeeding A of 245 will tally in the same cell. Cell T+126, in our example, will get to 26.

After all N values have been analyzed, the T Table will contain all the results in order. The subroutine PRINT will then output the tabulated results.

Many computing systems gratuitously zero out blocks of storage for their users (a practice much to be deplored). To insure that the CLEAR subroutine really works, then, two cells of block T are loaded with numbers that should later disappear.

H

STARTING N = 79100

245	245	100	100	33	139	126	126	139	139
139	126	126	126	126	77	126	126	126	126
126	126	77	77	126	126	126	126	126	126
126	307	77	77	77	77	126	139	77	77
77	77	77	126	126	82	126	77	126	126
126	77								

This is your test data for the 3X+1 problem. The 52 numbers are the A values for the 52 consecutive values of N starting, in this case, with 79100. Your program should produce this same information, but in tabulated form. Prepare for your test run(s) by tabulating this data now.

Personal Computing Software

When it became apparent in 1978 that personal computers were going to sell in large numbers, a peripheral industry--packaged programs for these machines--sprang up. The predictions at that time were that:

(a) The market was unlimited;

(b) People were crying for programs and would pay anything for them;

(c) Programs didn't have to be neat and certainly needed no documentation--just as long as they worked;

(d) Royalties were almost impossible to predict because they would be so large; and

(e) The whole business would be run honestly, efficiently, and in a gentlemanly fashion.

All the people who made those predictions are now a lot older and wiser. It came as a big surprise to many people that programs could be copied, and they were. They were not only copied for use by friends, but some scoundrels even copied programs from one software vendor for submission to another vendor.

The early programs ran heavily to games: endless rehashes of Tic-Tac-Toe, Hangman, Star Trek, and Bombs Away. Then we had the deluge of guess-a-number programs, and programs to facilitate drawing of Lissajou curves, followed shortly by simulations of many casino gambling games.

Finally programs began to be written and marketed to do useful, complex, and sophisticated things: medical records (in fact, all kinds of involved record keeping); all the standard problems of a small business; text editing and word processing; and statistical calculations.

The customers will no longer buy shoddy programming; the quality standards have risen an order of magnitude or so just in the last six months. The pirating problem is still with us, but is not so severe--no one knows why. One possible explanation is that the level of complexity of packaged programs has risen significantly, and hence the pirates no longer understand what they're pirating, and hence give up.

I started out to list software packages that might be needed and marketable, and to this end, I solicited help from several people.

Associate Editor David Babcock wrote:

> First of all, I think your question of what software is needed is premature. I think that the first question to be answered is what kind of software is needed. The obvious answer is QUALITY. But I think it goes deeper than that.
>
> The key word is personal. My Apple is my personal machine. I can do with it what I want in the way I want. Even if I was handed well documented, quality software I would change it. And with good reason: my machine is going to be different from every other machine. That's a big reason why there are so many different versions of the same program around. The software authors may brag that their XYZ program has a super duper option that the others don't have, but what I hear them saying is "I've personalized my software to me." It's interesting to me that these same people don't understand and indeed feel hurt when others change their software. Of course all of this isn't new; people have been customizing their cars, homes, etc., for years. What is different with the computer is that it is easy and inexpensive to do this personalizing.
>
> All of this leads me to some guidelines for software:
>
> 1. It must be relatively inexpensive.
> 2. It must be well written and well documented.
> 3. Sources must be supplied.
> 4. Programs should be heavily parameterized so that they can be easily changed. Specific information on how to change the program must be supplied.
>
> The above items go against the grain of professional software houses. They have a large investment in developing the software that must be protected. I'm not sure how to reconcile the conflicts, but I feel that the only way to get good software in the hands of the personal computer users is to follow guidelines like those I've listed.

Or, these remarks from Larry Clark:

> Most of my needs/wants fall into the category of "what Tandy forgot." Having a bit of Scotch heritage, I am reluctant to buy what I can create for myself free. However, in some cases the effort exceeds the cost.
>
> I have considered buying a package to renumber BASIC statements, but I feel that I can write such a program. My effort would be reduced if I had a listing of the TRS-80 internals. Hence I'm trying to decide whether to buy or build a disassembler. In general, I hate to buy anything that comes without source, since I fear there will be some minor but frustrating twiddle needed.
>
> High precision arithmetic wouldn't do much for me; I do not have the interest in numerical methods that you do. System software is what interests me primarily. If I thought there was a really good, disk-based editor, I'd buy it. Until then, I'm planning to grow my own, time permitting. The computer stores are crying for good business software, and quite a lot of it is starting to appear, quality unknown.

What really triggered me, though, was this appeal from Dr. Clifton Howard of Harrington Park, New Jersey:

> I bought an Apple II last year in the hope that it would bring order out of chaos in the collected documentation I have of several thousand ancestors. I would like to be able to store, file, sort, retrieve, and cross-reference genealogical data. I would like to be able to have pedigree, individual and family group printouts as well as indexes. Excellent work in this area has been done by the LDS church, but they use IBM 370s. Some work out of the University of Utah has focussed on minis using an excellent soundex code with pointer systems for parents and progeny, but the adaptation to microcomputers is not clear. This work has relevance to tracing genetic disorders, and there are other analogs. If others with the same interest come forward, a network of information and knowledge could be pooled and shared.

The following suggestions (by Paul Lamar) appeared in Kilobaud Microcomputing, July, 1979:

> We suggest you don't buy software that does any of the following:
>
> 1. Executes automatically after loading.
> 2. Modifies the screen memory while loading.
> 3. Cannot be loaded from disk using the BASIC DOS commands.
> 4. Cannot be unlocked using the BASIC DOS commands.
> 5. Cannot be listed.
> 6. Cannot be changed.
> 7. Has BASIC line numbers greater than 32000.

With all these viewpoints in mind, it might still make sense to list some needed software.

1. Good demonstration programs. The person who has just spent $1000 on a personal machine will meet--rather quickly--the question "Hey, what can it do?" What is needed here is a tape of half a dozen intelligent programs, such as: a complex pattern generator; some high-powered mathematics program (e.g., factor any given 8-digit number); a non-trivial game (e.g., Reversi); a simulation (e.g., of a roulette wheel); a program to calculate the distance between any two of a list of 30 cities, and between any one of those cities and any other point on earth, specified by latitude and longitude; a program to calculate the elapsed time in days between any two dates in the Christian era (this one would be quite difficult to write); and so on.

The calculation of the day of the week, given the calendar date, is a commonly seen program, and if repackaged as a demonstation it could be spectacular. At the end of this article it is shown how that might be done.

2. A high precision arithmetic package. The best of the current packaged arithmetic systems (e.g., floating BASIC) run to 9 significant digits, above which they break out into scientific notation, but still retaining only 9 digits. There are many situations that demand higher precision (see "How High the Precision?" in our issue 52). What is needed is a package written in machine language (but controlled via an interpreter) that can handle floating numbers of from one to, say, 100 digits. Such a package would run fast at one digit, but very slowly when working with 100 digit numbers.

3. There has been surprising little use of animation in packaged software. The microprocessor-controlled games at the quarter arcades make extensive use of clever animation, so it obviously can be done. What little animation we have seen so far (in Pong-type games, Bombs Away, Breakout) is mostly badly done and quite imperfect (e.g., the ball clearly hits the paddle on the screen, but it registers as a miss). Even staid statistical programs could make good use of blinking, prompting, reverse video, highlighting, and various cursors. With the possible exception of some of the chess programs, I have not seen one program for sale that makes intelligent use of the screen capability of the machine.

4. At least once a month now someone reinvents random number generators again and proceeds to describe, ad nauseum, one or two of the 9 standard statistical test of randomness. Someone should write a program (in BASIC, of course) to apply all of those tests to any given generator.

5. We seem to be quite short of clever utility programs. For example, when working in assembly language, and things go awry (as they do), it would be nice to have handy a "How Did I Get Here?" search routine, to search all of storage for a jump to XXXX. As has been pointed out, we lack properly written tools to enable us to produce better programs.

6. The gambling simulations (blackjack, roulette, craps, etc.) should be re-done in a form to facilitate trying out new gambling "systems."

7. There must be an awful lot of idle CPU time in the country. Some of the owners of personal machines might appreciate a base load problem--a problem that goes on endlessly, for which each user could contribute results. The problems that have appeared in POPULAR COMPUTING over the years furnish an excellent source of such base load problems.

8. There are already available excellent programs for many machines to play chess, checkers, Kalah, bridge, backgammon, and Othello. The demand for such game programs must be high, and more should be developed. The game of Pasta (described in our issue 12) is a contender, as is Oware (a better version of Kalah--described in issue 55), and Go-Moku (described in issue 49). For that matter, no one to our knowledge has tried programming a coaching version of chess, to teach the game. This would be a program in which pertinent comments would be made after each (human) play, of the order of:

HE CAN TAKE THAT PIECE EASILY.

YOUR KING IS DANGEROUSLY EXPOSED.

THAT WAS PROBABLY YOUR BEST MOVE.

coupled with the capability of withdrawing a move, in order to try another one. Such a program might make it easy to learn to play chess.

DAY-OF-THE-WEEK

The calculation of the day of the week, given a calendar date, keeps reappearing in the personal computing magazines. It is usually presented as a novel and useful calculation, which is nonsense. For usefulness, the proper tool for the job is called a calendar. Long-term calendars can be found in dictionaries, almanacs, and even in telephone books. However, with some minor modifications, the day of the week calculation can be turned into a splendid demonstration.

The actual calculation is best done with the classic formula devised by the Rev. Zeller:

$$F \equiv \left\{ \left[2.6M - .2 \right] + K + D + \left[\frac{D}{4} \right] + \left[\frac{C}{4} \right] - 2C \right\} \bmod 7$$

in which F is the day of the week, given by the table:

0 = Sunday
1 = Monday
2 = Tuesday
3 = Wednesday
4 = Thursday
5 = Friday
6 = Saturday

M is the month, with March = 1, April = 2,... December = 10, and January and February are months 11 and 12 of the preceeding year.

K is the day of the month
C is the century number
D is the year in the century.

Thus for December 7, 1941 we have

M = 10
K = 7
C = 19
D = 41

The square brackets, as usual, denote "greatest integer in."

[In adding the terms in the congruence, the sum field should be initialized to some high multiple of 7, like 77, to avoid going negative on subtracting 2C. The calculation of mod 7 is most easily accomplished simply by subtracting 7s from the sum.]

For Pearl Harbor day we have, taking the terms of Zeller's congruence in order:

$$\begin{array}{r} +25 \\ +\ 7 \\ +41 \\ +10 \\ +\ 4 \\ -38 \\ \hline = 49 \end{array}$$

and 49 modulo 7 (the remainder on division) is 0. The table then gives the result, Sunday. It should be pointed out that the operation:

$$\left[\frac{D}{4}\right] = \left[\frac{41}{4}\right] = 10$$

is exactly how a computer normally does division. The programming (say, in integer BASIC) is rather trivial and execution time is fast. Too fast, in fact; the effect of the demonstration is weakened by the excessive speed of the machine. It is much more impressive if the program is written to call for input, such as:

8, 23, 1963

(for August 23, 1963. On input, the months are numbered in their normal order), and then pause momentarily (as the machine enters the mull mode) before displaying

THE DAY IS ... FRIDAY

But this can be made even more dramatic. When showing off the program, the first date chosen should be the current date, for which the answer is obvious. The second date, then, could be the same date one year hence, for which the person demonstrating glibly explains that the result, of course, will be one day later in the week (except if Leap Day intervenes, of course). Thus, the demonstration comes to the third date, which can be solicited from the viewers; what is needed is some other date for which the day of the week is known. For the older crowd, Pearl Harbor day pops up. For others, there may be a person who was born on Thanksgiving, or Easter, or who otherwise knows a good test case. If none is forthcoming from the audience, then the demonstrator should suddenly remember Pearl Harbor day. What the program produces is:

Input: 12, 7, 1941

Output: THE DAY IS ... WEDNESDAY

followed by about 4 seconds of delay, and then:

OOPS ... SUNDAY

This implies, of course, calculating the correct result on the third entry, changing it to a wrong result, and wasting 4 seconds of CPU time. The latter trick can be as simple as:

```
510 FØR I = 1 TØ 2500
520 NEXT I
```

The false result can be programmed for the third entry, or the input routine can be rigged to do it on any entry. For example, an extra space after the entry of the input date (unnoticed by the viewers) can be the signal to miscalculate.

This provides an entertaining demonstration. The person demonstrating should, of course, then explain exactly what went on; we have enough misconceptions about computers already.

If the program does not edit the input data, then it will accept a date like:

13, 43, 1979

and output, as it should:

THE DAY IS ... TUESDAY

This rounds out the demonstration, to show that output can be no better than the input.

Note: Zeller's formula is good only for dates after 1752, the year when the current Gregorian calendar was adopted throughout most of the world.